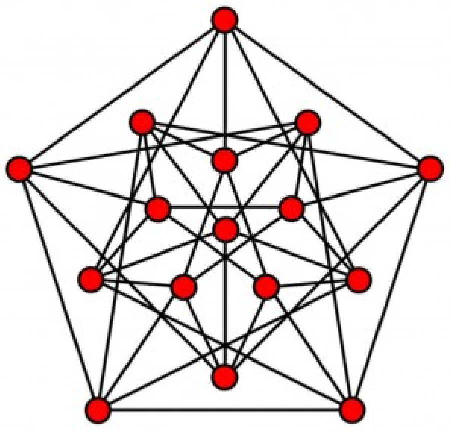
COMP 251 MT1 Notes

Algorithms and Data Structures

Written By

David Knapik

McGill University
david.knapik@mcgill.ca



February 26, 2018

Contents

1 Introduction - Good vs Bad Algorithms **2**

1.1 Software vs Hardware 2

2 Recursive Algorithms **2**

2.1 Divide and Conquer Algorithms 3

2.1.1 MergeSort 3

2.1.2 Binary Search 4

2.1.3 Master Theorem 5

2.2 Multiplication of Numbers 6

2.2.1 Primary School Long Multiplication 6

2.2.2 Russian Peasant Multiplication 6

2.2.3 Divide and Conquer Multiplication 6

2.3 Multiplication of Matrices 7

2.3.1 High School Matrix Multiplication 7

2.3.2 Divide and Conquer Matrix Multiplication 7

2.4 Fast Exponentiation 7

2.5 An Aside on Solving Recurrences 8

2.5.1 Domain Transformation 8

2.6 The Selection Problem 8

2.6.1 Motivation - The Median Problem 8

2.6.2 Naive Selection Algorithm 8

2.6.3 Randomized Selection Algorithm 9

2.6.4 Deterministic Selection Algorithm 10

2.7 Finding the Closest Pair of Points in the Plane 12

2.7.1 Exhaustive Search 12

2.7.2 1D Case and a Naive Approach 12

2.7.3 Divide and Conquer Algorithm with a Trick 12

1 Introduction - Good vs Bad Algorithms

Theorem 1.0.1 (Central Paradigm of CS)**.** *An algorithm $\mathcal{A}$ is good if $\mathcal{A}$ runs in **polynomial time** in the input size n. In other words, $\mathcal{A}$ runs in time $T(n) = O(n^k)$ for some constant k.*

Remark. "$\mathcal{A}$ runs in polynomial time in the input size n" is equivalent to "the input sizes that $\mathcal{A}$ can solve, in a fixed amount T of time, scales multiplicatively with increasing computational power".

We say that an algorithm is bad if it runs in exponential time.

Example 1.0.1. $T(n) = 2^n + 100n^5$ is bad.

Example 1.0.2 (Mergesort)**.** Mergesort runs in time $O(n \log n)$, so it is good.

Example 1.0.3 (BruteForce Search)**.** BruteForce Search runs in time $O(n \cdot n!) >> 2^n$, so it is bad.

This measure of quality or "goodness" is **robust**. All reasonable models of algorithms are polynomial time equivalent. Otherwise, one model could perform, say, an exponential number of operations in the time another model took to perform just one.

1.1 Software vs Hardware

Improvements in hardware will never overcome bad algorithm design. Indeed, current dramatic breakthroughs in CS are based upon better (faster and higher performance) algorithm techniques.

2 Recursive Algorithms

Solving a problem by reducing it (or a sub-problem of it) to another problem is the most fundamental technique in algorithm design. Specifically, algorithm $\mathcal{A}$ may use another algorithm $\mathcal{B}$ as a sub-routine. This has advantages:

1. Code verification: the correctness of $\mathcal{A}$ is independent of $\mathcal{B}$.
2. Code reuse: a great time-saver

A special case of this idea is when an algorithm calls itself $\rightarrow$ **recursion**.

2.1 Divide and Conquer Algorithms

Definition 2.1.1. A divide and conquer algorithm with parameters a, b, d is an algorithm which recursively breaks up a problem of size n into smaller subproblems such that:

1. $\exists$ exactly a subproblems.
2. Each subproblem has size at most $\frac{1}{b}n$.
3. Once solved, the solutions to the subproblems can be combined to produce a solution to the original problem in time $O(n^d)$.

So, the run time of a divide and conquer algorithm satisfies the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

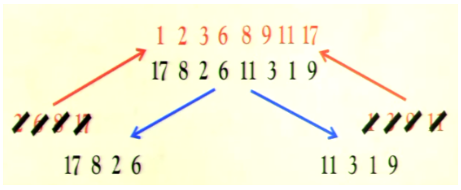
2.1.1 MergeSort

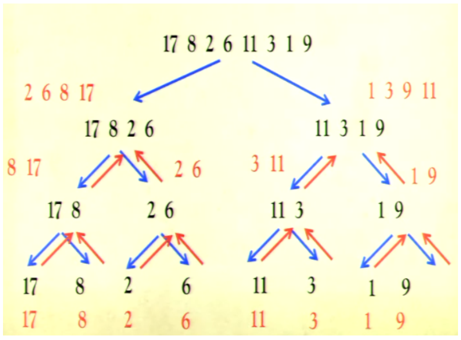
Here, we want to sort n numbers into non-decreasing order.

Algorithm 2.1.1. MergeSort$(x_1, ..., x_n)$

```
    **If** $n = 1$ **then** output $x_1$
    **Else** output Merge{MergeSort($x_1, ..., x_{\lfloor n/2 \rfloor}$),MergeSort($x_{\lfloor n/2 \rfloor+1}, ..., x_n$)}
```

Example 2.1.1.





Remark. MergeSort does work: The division process terminates with a set of base cases of size 1. Using strong induction, and given the validity of the Merge step, we are done. (MergeSort trivially works on the base cases).

Theorem 2.1.1. *MergeSort runs in time $O(n \cdot \log n)$.*

Proof. First, note that:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

where the first term is due to recursion on 2 problems with half the size and the second term is the linear time to merge 2 sorted lists. Also the base case is $T(1) = 1$.

Now, by adding **dummy numbers**, we may assume that n is a power of two; $n = 2^k$.

So, we have:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left(2T(n/4) + \frac{cn}{2}\right) + cn \\
&= 2^2 T\left(\frac{n}{4}\right) + 2cn \\
&= 2^3 T\left(\frac{n}{8}\right) + 3cn \\
&= ... \\
&= 2^k T(1) + kcn \\
&= 2^k + kcn \\
&= n(1 + kc)
\end{aligned}$$

So, $T(n) = O(nk) = O(n \log n)$. □

2.1.2 Binary Search

We can search for a key k in a sorted array list of cardinality n by using the binary search algorithm.

Algorithm 2.1.2. BinarySearch($a_1, ..., a_n : k$)

```
    While $n > 0$ do:
    if $a_{\lceil n/2 \rceil} = k$ output YES
    Else if $a_{\lceil n/2 \rceil} > k$ output BinarySearch($a_1, ..., a_{\lceil n/2 \rceil - 1} : k$)
    Else if $a_{\lceil n/2 \rceil} < k$ output BinarySearch($a_{\lceil n/2 \rceil + 1}, ..., a_{n-1}, a_n : k$)
    Output NO
```

Example 2.1.2. See many online or the slides to understand how it works.

Remark. BinarySearch does work: follows by strong induction.

Theorem 2.1.2. *BinarySearch runs in time $O(\log n)$.*

Proof. First, note that:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

where the first term is due to the recursion on one problem with half the size and the second term is the constant amount of additional work required. Also, the base case is $T(1) = 1$.

Now, by adding dummy numbers, we may assume that n is a power of two; $n = 2^k$.

So, we have:

$$\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c \\
&= (T(n/4) + c) + c \\
&= T\left(\frac{n}{4}\right) + 2c \\
&= T\left(\frac{n}{8}\right) + 3c \\
&= ... \\
&= T\left(\frac{n}{2^k}\right) + kc \\
&= 1 + kc \\
&= 1 + \log n \cdot c
\end{aligned}$$

Thus, $T(n) = O(\log n)$. □

2.1.3 Master Theorem

Theorem 2.1.3 (Master Theorem)**.** *If* $T(n) = aT(\frac{n}{b}) + O(n^d)$ *for constants* $a > 0, b > 1, d \geq 0$ *then:*

$$T(n) = \begin{cases} O(n^d) & a < b^d \quad \textit{Case I} \\ O(n^d \log n) & a = b^d \quad \textit{Case II} \\ O(n^{\log_b a}) & a > b^d \quad \textit{Case III} \end{cases}$$

Proof. Assume that n is a power of b: $n = b^l$. So,

$$\begin{aligned} T(n) &= n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + ... + a^l\left(\frac{n}{b^l}\right)^d \\ &= n^d\left(1 + a\left(\frac{1}{b}\right)^d + a^2\left(\frac{1}{b^2}\right)^d + ... + a^l\left(\frac{1}{b^l}\right)^d\right) \\ &= n^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + ... + \left(\frac{a}{b^d}\right)^l\right) \end{aligned}$$

■ Case I: $\frac{a}{b^d} < 1$. Let $\tau = \frac{a}{b^d}$. Then,

$$\begin{aligned} T(n) &= n^d \sum_{k=0}^{l} \tau^k \\ &= n^d\left(\frac{1-\tau^{l+1}}{1-\tau}\right) \quad \text{(using geometric series)} \\ &\leq n^d\left(\frac{1}{1-\tau}\right) \\ &= n^d\left(\frac{b^d}{b^d - a}\right) \end{aligned}$$

So, $T(n) = O(n^d)$.

■ Case II: $\frac{a}{b^d} = 1$. Then, $T(n) = n^d(l+1) = n^d(\log_b n + 1)$. So, $T(n) = O(n^d \log n)$.

■ Case III: $\frac{a}{b^d} > 1$. Let $\tau = \frac{a}{b^d}$. Then,

$$\begin{aligned} T(n) &= n^d \sum_{k=0}^{l} \tau^k \\ &= n^d\left(\frac{\tau^{l+1} - 1}{\tau - 1}\right) \\ &\leq n^d\left(\frac{\tau^{l+1}}{\tau - 1}\right) \end{aligned}$$

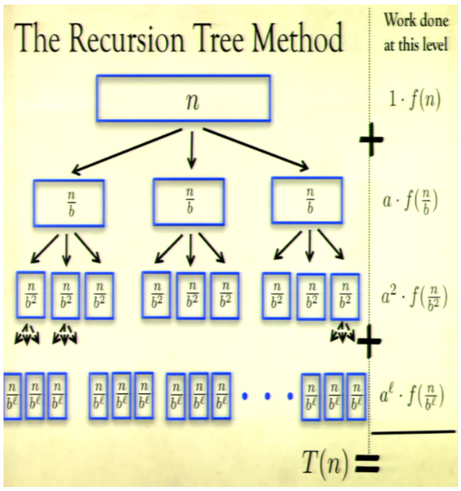
Thus,

$$\begin{aligned} T(n) &= O(n^d \tau^l) \\ &= O\left(n^d\left(\frac{a}{b^d}\right)^l\right) \\ &= O\left(\left(\frac{n}{b^l}\right)^d a^l\right) \\ &= O(a^l) \\ &= O(a^{\log_b n}) \\ &= O(n^{log_b a}) \end{aligned}$$

where in the last line we have used that $x^{\log_b y} = y^{\log_b x}$ for any base b. □

The ideas in the proof of the theorem are more important than the statement itself. Thus it is very important to understand the proof, which will then allow us to solve more general problems.

We note that the Master Theorem is a special case of the **Recursion Tree Method**. We illustrate the idea of the Recursion Tree Method using the model of the divide and conquer recursive formula:



Note that there are a^δ nodes at depth δ in the tree. This visualization allows us to slearly see what is happening in the proof of the Master Theorem.

2.2 Multiplication of Numbers

How long does it take to multiply 2 n-digit numbers?

2.2.1 Primary School Long Multiplication

This very familiar algorithm has run time $= \Omega(n^2)$.

2.2.2 Russian Peasant Multiplication

The algorithm is:

Algorithm 2.2.1. Mult(x,y)

```
    **If** $x = 1$ **then** output $y$
    **If** $x$ is odd **then** output $y+$ Mult$(\lfloor x/2 \rfloor, 2y)$
    **If** $x$ is even **then** output Mult$(x/2, 2y)$
```

This algorithm has a run time of $O(n^2)$.

2.2.3 Divide and Conquer Multiplication

Our goal is to have an algorithm to multiply 2 n-digit numbers which has runtime better than $O(n^2)$.

We have $\mathbf{x} = \underbrace{x_n x_{n-1} \ldots x_{\frac{n}{2}+1}}_{\mathbf{x_L}} \underbrace{x_{n/2} \ldots x_1}_{\mathbf{x_R}}$. Do the same for $\mathbf{y}$. Note that we have $\mathbf{x} = 10^{\frac{n}{2}} \mathbf{x_L} + \mathbf{x_R}$ and similarly for $\mathbf{y}$. Now,

$$\mathbf{xy} = 10^n \mathbf{x_L y_L} + 10^{n/2}(\mathbf{x_L y_R} + \mathbf{x_R y_L}) + \mathbf{x_R y_R}$$

So, we have 4 products with $n/2$ digit numbers. Thus $T(n) = 4T(n/2) + O(n)$. By the padding argument, we may assume n is a power of 2. Using the Master Theorem ($a = 4, b = 2, d = 1$, so we have Case III), we get a runtime of $O(n^{\log_2 4}) = O(n^2)$. Not good. But, there is a trick we can do which originates from Gauss multiplying complex numbers:

The Trick

Note that $(a+bi)(c+di) = ac - bd + (bc+ad)i$. This seems to require 4 products. However, $(bc+ad) = (a+b)(c+d) - ac - bd$, so we can calculate ac and bd and then only need to perform one more product, $(a+b)(c+d)$. In our context, we let $i := 10^{n/2}$. Then,

$$(\mathbf{x_L y_R} + \mathbf{x_R y_L}) = \mathbf{x_R y_R} + \mathbf{x_L y_L} - (\mathbf{x_R} - \mathbf{x_L})(\mathbf{y_R} - \mathbf{y_L})$$

So, we end up with 3 products of $n/2$ digit numbers. Thus, $T(n) = 3T(n/2) + O(n)$. Applying the Master Theorem, we get that the runtime is $O(n^{\log_2 3}) = O(n^{1.59})$.

Remark. We can actually multiply 2 n-digit numbers in $O(n \log n)$ by using FFTs.

2.3 Multiplication of Matrices

How long does it take to multiply 2 $n \times n$ matrices?

2.3.1 High School Matrix Multiplication

This familiar algorithm has a runtime of $\Omega(n^3)$. Can we do better?

2.3.2 Divide and Conquer Matrix Multiplication

Have $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$ where the entries are submatrices themselves. Then,

$$Z = XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

(verify this for yourself). So, we have 8 products of $n/2 \times n/2$ matrices. Thus $T(n) = 8T(n/2) + O(n^2)$ and by the Master Theorem, the runtime is $O(n^{\log_2 8}) = O(n^3)$.

The Trick

Claim (exercise) that

$$Z = XY = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

where $S_1 = (B-D)(G+H), S_2 = (A+D)(E+H), S_3 = (A-C)(E+F), S_4 = (A+B)H, S_5 = A(F-H), S_6 = D(G-E), S_7 = (C+D)E$.

So, then we have only 7 products of $n/2 \times n/2$ matrices. This results in $T(n) = 7T(n/2) + O(n^2)$ and thus by Master Theorem the runtime is $O(n^{\log_2 7}) = O(n^{2.81})$.

Remark. There is actually an algorithm that can do $O(n^{2.37})$.

2.4 Fast Exponentiation

The algorithm is:

Algorithm 2.4.1. FastExp(x, n)

```
    **If** $n = 1$ **then** output $x$
    **Else**
    **If** $n$ is even **then** output FastExp$(x, \lfloor n/2 \rfloor)^2$
    **If** $n$ is odd **then** output $x$FastExp$(x, \lfloor n/2 \rfloor)^2$
```

We have that $T(n) \le T(\lfloor n/2 \rfloor) + 2$, so $T(n) = T(n/2) + O(1)$ and thus by the Master Theorem the runtime is $O(n^0 \log n) = O(\log n)$.

2.5 An Aside on Solving Recurrences

MergeSort actually has the recurrence $\hat{T}(n) = \hat{T}(\lfloor n/2 \rfloor) + \hat{T}(\lceil n/2 \rceil) + cn$. We had used dummy entries (we found $\bar{n}$, the smallest power of 2 greater than n). $\hat{T}(n) \le T(\bar{n}) = O(\bar{n} \log \bar{n}) = O(n \log n)$.

2.5.1 Domain Transformation

We have an alternative solution by using a Domain Transformation. Let $T(n) = \hat{T}(n+2)$. Then:

$$\begin{aligned}
\hat{T}(n) &\le \hat{T}\left(\frac{n+2}{2} + 1\right) + c(n+2) \\
&\le \hat{T}\left(\frac{n+2}{2} + 1\right) + \hat{c}n \\
&= \hat{T}\left(\frac{n}{2} + 2\right) + \hat{c}n \\
&= T\left(\frac{n}{2}\right) + \hat{c}n
\end{aligned}$$

So $T(n) = O(n \log n)$ and therefore $\hat{T}(n) = T(n-2) = O(n \log n)$.

2.6 The Selection Problem

2.6.1 Motivation - The Median Problem

Suppose that we want to find the median of a set $\mathcal{S} = \{x_1, ..., x_n\}$. We could just sort the list and then output the $\lceil n/2 \rceil$-th number. Using MergeSort (or otherwise), this would take $O(n \log n)$. Can we do this any faster? Well, to answer this question, it is more convenient to study the more general problem called the selection problem.

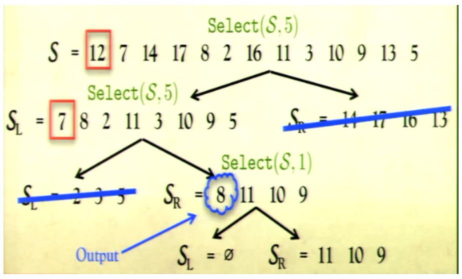
2.6.2 Naive Selection Algorithm

So, we would like to find the k-th smallest number in $\mathcal{S}$.

Algorithm 2.6.1. Select($\mathcal{S}, k$)

```
    **If** $|\mathcal{S}| = 1$ **then** output $x_1$
    **Else**
    set $\mathcal{S}_L = \{x_i \in \mathcal{S} : x_i < x_1\}$ set $\mathcal{S}_R = \{x_i \in \mathcal{S} \setminus x_1 : x_i \ge x_1\}$
    **If** $|\mathcal{S}_L| = k - 1$ **then** output $x_1$
    **If** $|\mathcal{S}_L| > k - 1$ **then** output Select($\mathcal{S}_L, k$)
    **If** $|\mathcal{S}_L| < k - 1$ **then** output Select($\mathcal{S}_R, k - 1 - |\mathcal{S}_L|$)
```

Example 2.6.1.



What is the runtime? Well,

$$\begin{aligned} T(n) &= n-1+T(\max\{|\mathcal{S}_L||\mathcal{S}_R|\}) \\ &= n-1+T(n-1) \\ &= (n-1)+(n-2)+T(n-2) \\ &= \ldots \\ &= (n-1)+(n-2)+(n-3)+\ldots+2+1 \\ &= \frac{1}{2}n(n-1) \end{aligned}$$

so $T(n)=\Omega(n^2)$. This is terrible and needs to be fixed by having a better choice for the pivot.

Balanced Pivots

The problem with the above algorithm is that it repeatedly pivots on the first number in the current list. If we are unlucky, this pivot could be very "unbalanced".What we mean by this is that $\max\{|\mathcal{S}_L|,|\mathcal{S}_R|\}\approx n$ and $\min\{|\mathcal{S}_L|,|\mathcal{S}_R|\}\approx 0$ so that the subproblems have very different sizes. We would like to pick a pivot that is "balanced", in the sense that $\max\{|\mathcal{S}_L|,|\mathcal{S}_R|\}\approx n/2$ and $\min\{|\mathcal{S}_L|,|\mathcal{S}_R|\}\approx n/2$. We cant really guarantee exactly that, but we can come close.

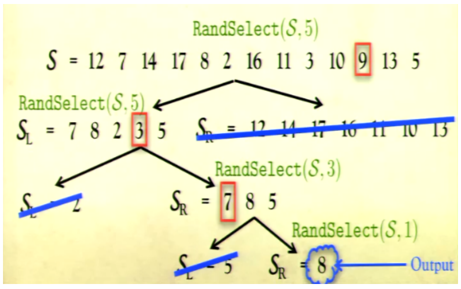
2.6.3 Randomized Selection Algorithm

Here, we just choose the pivot at random (uniformly) from $\mathcal{S}$.

Algorithm 2.6.2. RandSelect($\mathcal{S},k$)

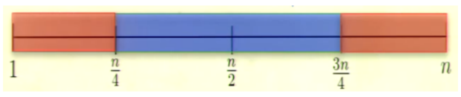
```
    **If** $|\mathcal{S}|=1$ **then** output $x_1$
    **Else** Pick a random pivot $x_\tau\in\{x_1,...,x_n\}$
    set $\mathcal{S}_L=\{x_i\in\mathcal{S}:x_i<x_\tau\}$
    set $\mathcal{S}_R=\{x_i\in\mathcal{S}\setminus x_1:x_i\geq x_\tau\}$
    **If** $|\mathcal{S}_L|=k-1$ **then** output $x_\tau$
    **If** $|\mathcal{S}_L|>k-1$ **then** output Select($\mathcal{S}_L,k$)
    **If** $|\mathcal{S}_L|<k-1$ **then** output Select($\mathcal{S}_R,k-1-|\mathcal{S}_L|$)
```

Example 2.6.2.



Good vs Bad Pivots

Imagine all the numbers are in sorted order.



With $\mathbb{P}=\frac{1}{2}$ the pivot x_τ lies between the first and third quartiles. In this case we say that such a pivot is good. Otherwise, the pivot is bad. If the pivot is good, note that then $\max\{|\mathcal{S}_L|,|\mathcal{S}_R|\}\leq\frac{3}{4}n$.

For randomized algorithms, we are interested in the **expected runtime** $\bar{T}(n):=\mathbb{E}(T(n))$, NOT the worst case runtime.

We have that:

$$\bar{T}(n) \le \frac{1}{2}\bar{T}\left(\frac{3n}{4}\right) + \frac{1}{2}\bar{T}(n) + O(n)$$

where the first term is due to the probability of a good pivot and the second term is due to the probability of a bad pivot. So rearranging, we obtain $\frac{1}{2}\bar{T}(n) \le \frac{1}{2}\bar{T}(\frac{3n}{4}) + O(n) \ \rightarrow\ \bar{T}(n) \le \bar{T}(\frac{3n}{4}) + O(n)$. And therefore by the Master Theorem, we have $\bar{T}(n) = O(n)$.

2.6.4 Deterministic Selection Algorithm

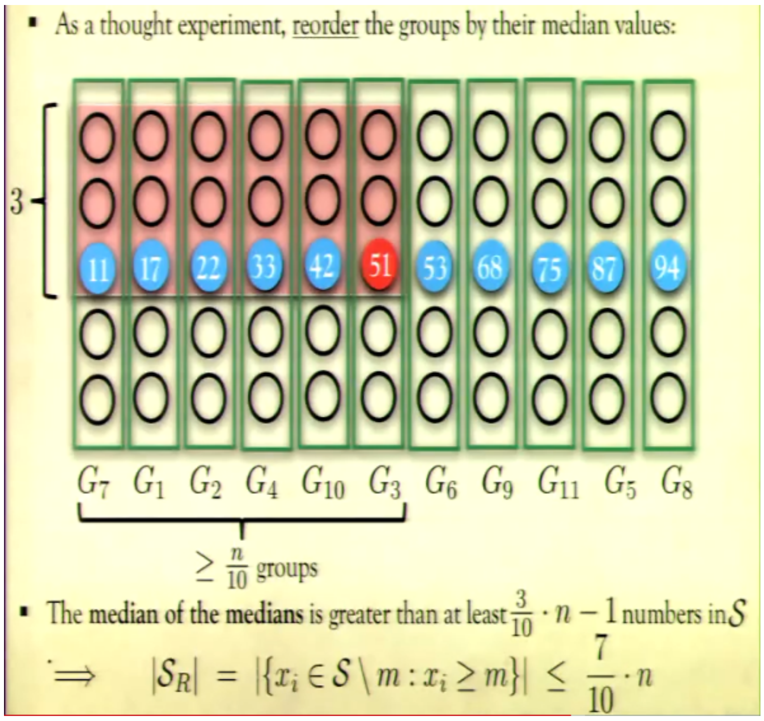
Can we have a linear time algorithm for the Selection Problem in the deterministic case? Well, this boils down to requiring a deterministic method to find a good pivot. The idea here is the Median of the Medians.

Median of the Medians

Divide $\mathcal{S}$ into groups of cardinality 5:

$$G_1 = \{x_1, ..., x_5\},\ G_2 = \{x_6, ..., x_{10}\}, ..., G_{n/5} = \{x_{n-4}, ..., x_n\}$$

Now, sort each group and let z_i be the median of the group G_i. Let m be the median of $\mathcal{Z} := \{z_1, ..., z_{n/5}\}$. Reorder the groups by their median values.

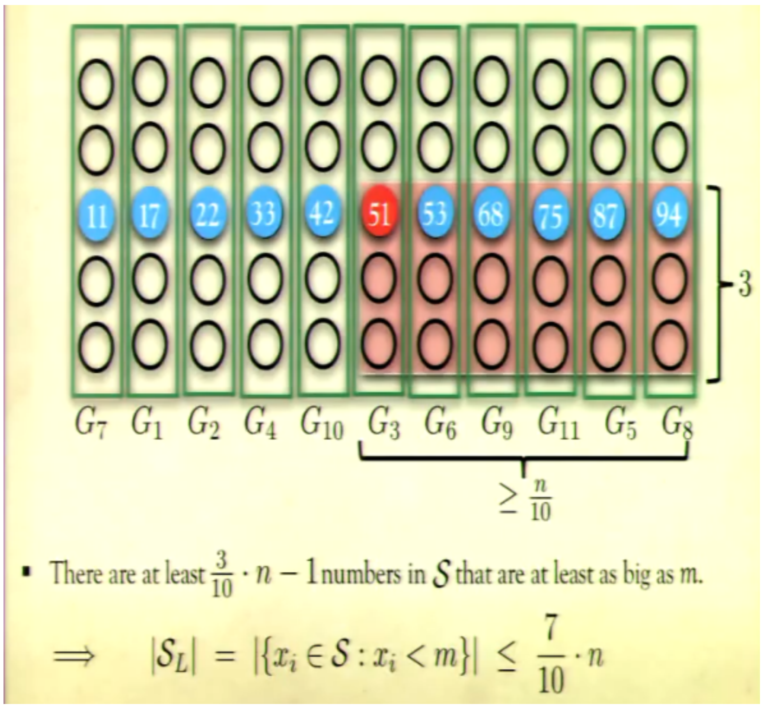


- As a thought experiment, reorder the groups by their median values:

3
11
17
22
33
42
51
53
68
75
87
94
G_7 G_1 G_2 G_4 G_{10} G_3 G_6 G_9 G_{11} G_5 G_8
$\ge \frac{n}{10}$ groups

- The median of the medians is greater than at least $\frac{3}{10}\cdot n - 1$ numbers in $\mathcal{S}$

$$\implies \quad |\mathcal{S}_R| = |\{x_i \in \mathcal{S} \setminus m : x_i \ge m\}| \le \frac{7}{10}\cdot n$$


11
17
22
33
42
51
53
68
75
87
94
3
G_7 G_1 G_2 G_4 G_{10} G_3 G_6 G_9 G_{11} G_5 G_8
$\geq \frac{n}{10}$

- There are at least $\frac{3}{10} \cdot n - 1$ numbers in $\mathcal{S}$ that are at least as big as m.

$$\Longrightarrow \quad |\mathcal{S}_L| = |\{x_i \in \mathcal{S} : x_i < m\}| \leq \frac{7}{10} \cdot n$$

So, using m as a pivot, we have: $\max\{|\mathcal{S}_L|, |\mathcal{S}_R|\} \leq \frac{7}{10}n$. So, the median of the medians is a good pivot. But, how do we actually find the median of the medians? Well, we just use the same deterministic algorithm:

Algorithm 2.6.3. DetSelect($\mathcal{S}, k$)

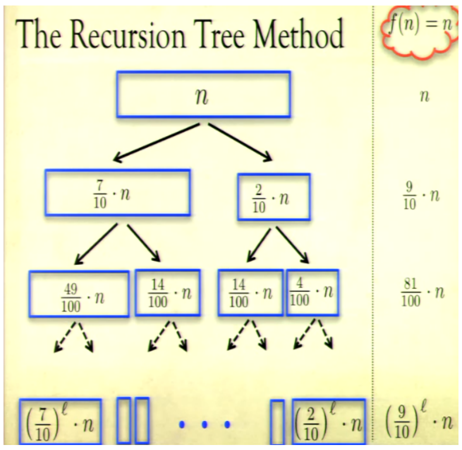
```
    If $|\mathcal{S}| = 1$ then output $x_1$
    Else
    Partition $\mathcal{S}$ into $\lceil n/5 \rceil$ groups of 5.
    For $j = \{1, 2, ..., \lceil n/5 \rceil\}$
    Let $z_j$ be the median of group $G_j$
    Let $\mathcal{Z} = \{z_1, ..., z_{\lceil n/5 \rceil}\}$
    Set $m \leftarrow$ DetSelect($\mathcal{Z}, \lceil n/10 \rceil$)
    Set $\mathcal{S}_L = \{x_i \in \mathcal{S} : x_i < m\}$
    Set $\mathcal{S}_R = \{x_i \in \mathcal{S} \setminus m : x_i \geq m\}$
    If $|\mathcal{S}_L| = k - 1$ then output $m$
    If $|\mathcal{S}_L| > k - 1$ then output DetSelect($\mathcal{S}_L, k$)
    If $|\mathcal{S}_L| < k - 1$ then output DetSelect($S_R, k - 1 - |\mathcal{S}_L|$)
```

We have:

$$T(n) \leq \left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + O(n)$$

The first term is due to pivoting on the median of the medians giving a smaller sub problem. The second term is from finding the median of the medians. The last term is for breaking into groups of size 5, finding the median of each group, and pivoting on the median of medians.

Note that here, we cannot apply Master Theorem. But, we can use the ideas in the proof of the Master Theorem i.e. the Recursion Tree Method:



This yields $T(n) = O(n)$.

2.7 Finding the Closest Pair of Points in the Plane

Given n points $\mathcal{P} = \{\mathbf{p_1}, ..., \mathbf{p_n}\}$ in the XY plane, we would like to find the closest pair of points.

2.7.1 Exhaustive Search

The first simplest algorithm to try is Exhaustive Search. The idea is to calculate the distance between every pair of points and then output the pair with the shortest pairwise distance. Since there are n points, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of points and thus the runtime is $O(n^2)$. Is there anything faster?

2.7.2 1D Case and a Naive Approach

It will be informative to first examine our problem in one dimension.

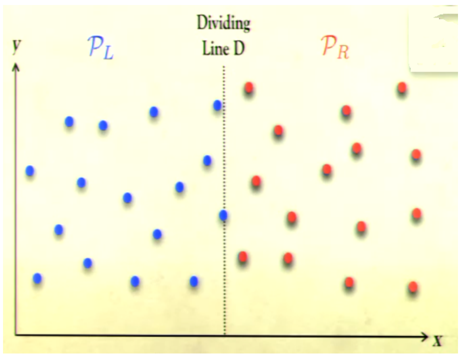
In 1 dimension, the closest pair of points must be adjacent in their $x-$ordering. Thus, we need to calculate only $n-1$ pairwise distances, resulting in a runtime of $O(n)$ if the points are sorted in the x coordinate and runtime of $O(n \log n + n) = O(n \log n)$ otherwise.

What happens if we apply this idea in 2 dimensions? Well lets order the points by their x coordinate (called the x- ordering) and then order the points by their y coordinate (called the y- ordering). Then, find the pair of points closest in their $x-$ coordinate. And, find the pair of points closest in their $y-$ coordinate. From these 2 pairs, output the pair that are closest together. Obviously, this algorithm does not work at all.

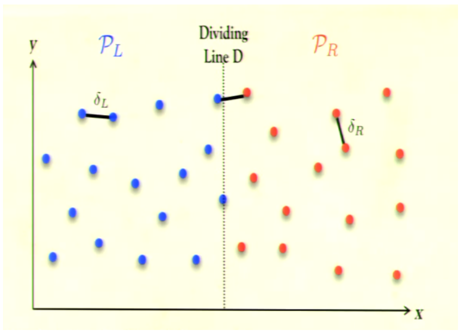
2.7.3 Divide and Conquer Algorithm with a Trick

In this subsection, we build up the algorithm to find the closest pair of points in the plane.

First, partition the points into 2 groups of cardinality $n/2$. We do this via the $x-$ ordering , using a dividing line D. Select D to pass through the point with the median x coordinate.



Note that we can find the median from the previous section in $O(n)$. Now, recursively search for the closest pairs in $\mathcal{P}_L$ and $\mathcal{P}_R$. BUT! closest pair could have one point in $\mathcal{P}_L$ and the other in $\mathcal{P}_R$.



Thus, after calculating δ_L and δ_R, we must check that there is no better solution between a point in $\mathcal{P}_L$ and a point in $\mathcal{P}_R$.

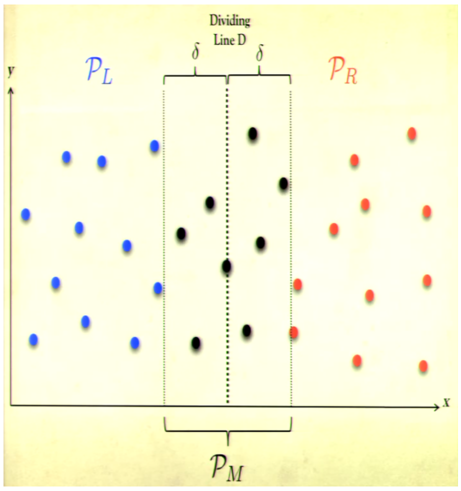
So, up to this point, our algorithm is:

1. Find the point $\mathbf{q}$ with the median x coordinate.
2. Partition $\mathcal{P}$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ by using $\mathbf{q}$.
3. Recursively find the closest pairs of points in $\mathcal{P}_L$ and $\mathcal{P}_R$.
4. Find the closest pair with one point in $\mathcal{P}_L$ and the other point in $\mathcal{P}_R$.
5. Amongst the 3 pairs found, output the closest pair.

With this algorithm we have $T(n) = 2T(n/2) + O(n^2)$ which by the Master Theorem is $O(n^2)$. This is not good. The problem here is the bottleneck step, which is step number 4.

So, since it takes too long to measure the distance between every point in $\mathcal{P}_L$ and $\mathcal{P}_R$, we need a trick.

By solving the sub-problems of $\mathcal{P}_L$ and $\mathcal{P}_R$, we know that the minimum pairwise distance is at most $\delta = \min\{\delta_R, \delta_L\}$. So, lets use δ to reduce the number of distances we measure between $\mathcal{P}_L$ and $\mathcal{P}_R$. The key observation here is that to find a better solution than δ, the 2 points in $\mathcal{P}_L$ and $\mathcal{P}_R$ must be very close to the dividing line D.



More formally, we state:

Lemma 2.7.1. Let $\mathbf{p_i} \in \mathcal{P}_L$ and $\mathbf{p_j} \in \mathcal{P}_R$. If $d(\mathbf{p_i}, \mathbf{p_j}) \leq \delta$ then $\{\mathbf{p_i}, \mathbf{p_j}\} \subset \mathcal{P}_M$.

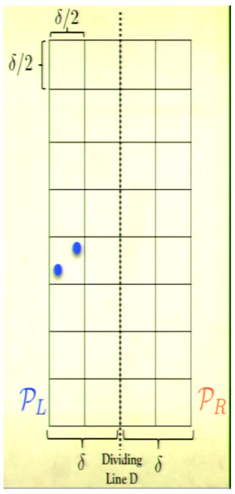
Proof. WLOG take $\mathbf{p_i} \notin \mathcal{P}_M$. Then, $d(\mathbf{p_i}, \mathbf{p_j}) > \delta$ $\square$

So, if the closest pair is not in one of the 2 subproblems, then it is in $\mathcal{P}_M$. So, our modified recursive algorithm so far is :

1. Find the point $\mathbf{q}$ with the median x coordinate.
2. Partition $\mathcal{P}$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ by using $\mathbf{q}$.
3. Recursively find the closest pairs of points in $\mathcal{P}_L$ and $\mathcal{P}_R$.
4. Find the closest pair of points in $\mathcal{P}_M$.
5. Amongst the 3 pairs found, output the closest pair.

Immediately, we notice an issue. $\mathcal{P}_M$ could contain all (or most) of the points. But in this case, then the point in $\mathcal{P}_M$ cover a narrow band of the x axis. So, we have a target distance of δ.

Consider the area of the plane within distance δ of the line D. Divide this area up into small squares of width $\delta/2$:



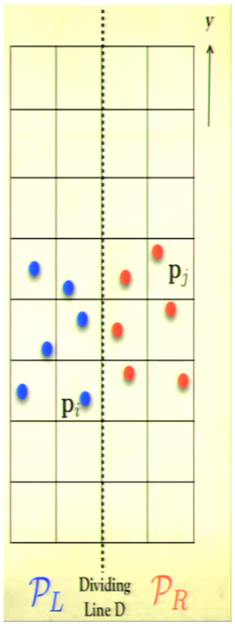
We claim that no two points of $\mathcal{P}$ lie in the same square.

Proof. WLOG take 2 points in a square in $\mathcal{P}_L$. Then, the pairwise distance is $\leq \sqrt{(\delta/2)^2 + (\delta/2)^2} = \frac{\delta}{\sqrt{2}} < \delta$. This contradicts the fact that the smallest pairwise distance in $\mathcal{P}_L$ is at least δ $\square$

Now,

Theorem 2.7.1. *Let $\mathbf{p_i} \in \mathcal{P}_L$ and $\mathbf{p_j} \in \mathcal{P}_R$. If $d(\mathbf{p_i}, \mathbf{p_j}) \leq \delta$ then $\mathbf{p_i}$ and $\mathbf{p_j}$ have at most 10 points between them in the y -ordering of $\mathcal{P}_M$.*

Proof. WLOG take $\mathbf{p_i}$ below $\mathbf{p_j}$ in the y order. Then, $\mathbf{p_j}$ is in the same row of squares as $\mathbf{p_i}$ or in the next 2 higher rows. (If not, then $d(\mathbf{p_i}, \mathbf{p_j}) > \delta$ leads to a contradiction). But, by our claim, we can have at most one point in each square.



So, $\exists$ at most 10 points between $\mathbf{p_i}$ and $\mathbf{p_j}$ in the y order of $\mathcal{P}_M$. $\square$

Now, the basic idea for the $1D$ algorithm will work here. To find the closest pair in $\mathcal{P}_M$ we first sort the points by the y coordinate. Then, rather than finding the distances between points that are one apart in the y order, we find the distances for all pairs up to 11 places apart. Thus, we calculate less than $11n$ pairwise distances. After doing so: either we find a pair of points at distance less than δ OR we conclude that no such pair of points exists.

So, given $\delta = \min\{\delta_L, \delta_R\}$, we can check if there exists a pair of points between $\mathcal{P}_L$ and $\mathcal{P}_R$ that are closer than δ in time $O(n)$. So, we finally have our algorithm:

1. Find the point $\mathbf{q}$ with the median x coordinate.
2. Partition $\mathcal{P}$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ by using $\mathbf{q}$.
3. Recursively find the closest pairs of points in $\mathcal{P}_L$ and $\mathcal{P}_R$.
4. Find the closest pair of points in $\mathcal{P}_M$ using our enhance 1D algorithm.
5. Amongst the 3 pairs found, output the closest pair.

The recursive formula for the running time is:

$$T(n) = 2T(n/2) + O(n)$$

where the $O(n)$ is for: finding the median with respect to x coordinates, partitioning into $\mathcal{P}_L$ and $\mathcal{P}_R$, finding $\mathcal{P}_M$, and applying 1D algorithm on $\mathcal{P}_M$. By the Master Theorem, this is $O(n \log n)$.

Acknowledgements

These notes have been transcribed from the lectures given by Prof. Adrian Vetta. Any figures are screenshots of the recorded lectures. This document is for personal use only.